# NETWORK PROGRAMMING IV - ASYNCHRONOUS REQUEST HANDLING: THREADINGMIXIN AND FORKINGMIXIN





## Note

This chapter is based on socketserver - A framework for network servers.

We'll see how we make **asynchronous request** using **ThreadingMixIn** and **ForkingMixIn**.

# Asynchronous Request Handling

In the previous chapter, we used TCPServer which process requests synchronously. That means each request must be completed before the next request can be started. This isn't suitable if each request takes a long time to complete, because it requires a lot of computation, or because it returns a lot of data which the client is slow to process.

The solution to this is to create a **separate process** or **thread** to handle each request. The **ForkingMixIn** and **ThreadingMixIn** mix-in classes can be used to support asynchronous behavior.

# Asynchronous Request handling Server code

```
# async.py

import socket
import threading
import SocketServer

class ThreadedTCPRequestHandler(SocketServer.BaseRequestHandler):

    def handle(self):
        data = str(self.request.recv(1024))
        cur_thread = threading.current_thread()
        response = bytes("{}: {}".format(cur_thread.name, data))
        self.request.sendall(response)

class ThreadedTCPServer(SocketServer.ThreadingMixIn, SocketServer.TCPServer):
    pass

def client(ip, port, message):
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect((ip, port))
    try:
        sock.sendall(bytes(message))
        response = str(sock.recv(1024))
        print("Received: {}".format(response))
    finally:
        sock.close()

if __name__ == "__main__":
    # port 0 means to select an arbitrary unused port
    HOST, PORT = "localhost", 0

    server = ThreadedTCPServer((HOST, PORT), ThreadedTCPRequestHandler)
    ip, port = server.server_address

    # start a thread with the server.
    # the thread will then start one more thread for each request.
    server_thread = threading.Thread(target=server.serve_forever)

    # exit the server thread when the main thread terminates
    server_thread.daemon = True
    server_thread.start()
    print("Server loop running in thread:", server_thread.name)

    client(ip, port, "Hello World 1")
    client(ip, port, "Hello World 2")
```

```
    client(ip, port, "Hello World 3")

    server.shutdown()
```

When inheriting from **ThreadingMixIn** for threaded connection behavior, we should explicitly declare how we want our threads to behave on an abrupt shutdown.
The **ThreadingMixIn** class defines an attribute **daemon_threads**, which indicates whether or not the server should wait for thread termination. We should set the flag explicitly if we would like threads to behave autonomously. The default value is False, meaning that Python will not exit until all threads created by **ThreadingMixIn** have exited. In the code, we set it **True**, which means Python will exit the server thread when the main thread terminates not waiting for other threads' exit.

Forking and threading **TCPServer** can be created using
the **ForkingMixIn** and **ThreadingMixIn** mix-in classes. For instance, a threading TCP server class is created as follows:

```
class ThreadingTCPServer(ThreadingMixIn, TCPServer): pass
```

The mix-in class must come first, since it overrides a method defined in TCPServer. Setting the various attributes also change the behavior of the underlying server mechanism.

To implement a service, we must derive a class from **BaseRequestHandler** and redefine its **handle()** method:

```
class ThreadedTCPRequestHandler(SocketServer.BaseRequestHandler):

    def handle(self):
        data = str(self.request.recv(1024))
        cur_thread = threading.current_thread()
        response = bytes("{}: {}".format(cur_thread.name, data))
        self.request.sendall(response)
```

The output from a run should look like this:

```
$ python async.py
('Server loop running in thread:', 'Thread-1')
Received: Thread-2: Hello World 1
Received: Thread-3: Hello World 2
Received: Thread-4: Hello World 3
```

# Python Network Programming

Network Programming - Server & Client A : Basics

Network Programming - Server & Client B : File Transfer

Network Programming II - Chat Server & Client

Network Programming III - SocketServer

Network Programming IV - SocketServer Asynchronous request